Functional Programming Paradigm

• The program is a collection of **functions**
  – A function computes and returns a value
  – No side-effects (i.e., no changes to state)
  – No program variables whose values change
    • Basically, no assignments
• Languages: LISP, Scheme (dialect of LISP from MIT, mid-70s), ML, Haskell, ...
• Functions as first-class entities
  – A function can be a **parameter** of another function
  – A function can be the **return value** of another function
  – A function could be an **element of a data structure**
  – A function can be created at run time
Outline

• **Language elements:**
  – Atoms and lists
• Evaluating expressions
  – Function application
  – Quoting an expression
  – Conditionals
  – Defining functions
• Examples
• S-expressions
• Function call semantics & higher-order functions
• More examples and features
Data Objects in Scheme

• **Atoms**
  – Numeric constants: 5, 20, -100, 2.788
  – Boolean constants: #t (true) and #f (false)
  – String constants: “hi there”
  – Character constants: \a
  – **Symbols**: f, x, +, *, null?, set!
    • Roughly speaking, equivalent to identifiers in imperative languages
  – Empty list: ( )

• **Lists**
  – (e₁ e₂ ... eₙ) where eᵢ is an atom or list
Examples of Lists

• (A B C)
• ((A B) C)
• ((3) (4) 5)
• (A B (C D))
• ((A))
• ()
• ()
Lists

• List elements can be atoms or other lists
  – ( ( 3 4 ) 5 ( 6 ) ) is a list with 3 elements
  – Thus, lists are heterogeneous: the elements do not have to be of the same type

• Empty list ( ) - has zero elements
  – Operations car and cdr are not defined for an empty list – run-time error
Lists

• **car** for a list produces the first element of the list (the list **head**)
  – e.g. for ( ( A B ) ( C D ) E ) will produce ( A B )

• **cdr** produces the **tail** of the list: a list containing all elements except the first
  – e.g. for ( ( A B ) ( C D ) E ) will produce ( ( C D ) E )

• **cons** adds to the beginning of the list
  – cons of A and ( B C ) is ( A B C )
  – e.g., cons of car of x and cdr of x is x
Outline

• Language elements:
  – Atoms and lists

• Evaluating expressions
  – Function application
  – Quoting an expression
  – Conditionals
  – Defining functions

• Examples

• S-expressions

• Function call semantics & higher-order functions

• More examples and features
Data vs. Code

• Interpreter for an imperative language: the input is code+data, the output is data (values)

• Everything in Scheme is an S-expression
  – The “program” we are executing is an S-expression
  – The intermediate values and the output values of the program are also S-expressions
    • Data and code are really the same thing

• Example: an expression that represents function application (i.e., function call) is a list \((f \; p1 \; p2 \; \ldots)\)
  – \(f\) is an S-expression representing the function we are calling; \(p1\) is an S-expression representing the first actual parameter, etc.
Using Scheme

• **Read**: you enter an expression
• **Eval**: the interpreter evaluates the expression
• **Print**: the interpreter prints the resulting value
• stdlinux: at the prompt, type `scheme48`

> *type your expression here*
the interpreter prints the value here

> ,help
> ,exit
Evaluation of Atoms

• Numeric constants, string constants, and character constants evaluate to themselves

> 4.5   > #t
 4.5    #t
> "This is a string"   > #f
"This is a string"    #f

• Symbols do not have values to start with
  – They may get “bound” to values, as discussed later
  > x
  Error: undefined variable x

• The empty list () does not have a defined value
Outline

• Language elements:
  – Atoms and lists

• Evaluating expressions
  – Function application
  – Quoting an expression
  – Conditionals
  – Defining functions

• Examples

• S-expressions

• Function call semantics & higher-order functions

• More examples and features
Function Application

• \((+ 5 6)\)
  - This S-expression is a “program”; here \(+\) is a symbol “bound” to the built-in function for addition
  - The evaluation by the interpreter produces the S-expression 11

• Function application: \((f \ p1 \ p2 \ ...)\)
  - The interpreter evaluates S-expressions \(f\), \(p1\), \(p2\), etc.
  - The interpreter invokes the resulting function on the resulting values
Outline

• Language elements:
  – Atoms and lists

• Evaluating expressions
  – Function application
  – Quoting an expression
  – Conditionals
  – Defining functions

• Examples

• S-expressions

• Function call semantics & higher-order functions

• More examples and features
Quoting an Expression

• When the interpreter sees a non-atom, it tries to evaluate it as if it were a function call
  – But for (5 6), what does it mean?
    • “Error: attempt to call a non-procedure”

• We can tell the interpreter to evaluate an expression to itself
  – (quote (5 6)) or simply '(5 6)
  – Evaluates to the S-expression (5 6)
  – The resulting expression is printed by the Scheme interpreter as '(5 6)
Examples

> (+ (+ 3 5) (car (7 8)))
Errors
1> Ctrl-D
> (+ (+ 3 5) (car '(7 8)))
15
> (car (7 10))
Errors
1> (car '(7 10))
7
1> (+ (car '(7 10)) (cdr '(7 10)))
Errors
2> (+ (car '(7 10)) (car (cdr '(7 10))))
17
More Examples

> (cons (car '(7 10)) (cdr '(7 10)))
'(7 10)

> a
Error

> (cdr '(A B))
'(b)

> (cons 'a '(b))
'(a b)

> (car '(A B))
'a

> (cons 'a 'b)
'(a . b)
More Examples

> (equal? #t #f)  > (equal? '() #f)
#f  #f

> (equal? #t #t)  > (equal? (+ 7 5) (+ 5 7))
#t  #t

> (equal? (cons 'a '(b)) '(a b))
#t

> (pair? '(7 . 10))  > (pair? 7)  > (pair? '())
#t  #f  #f

> (null? '())  > (null? #f)  > (null? '(b))
#t  #f  #f
More Examples

> (even? 7)  > (even? 8)  
#f   #t
> (even? (+ 7 7))  > (even? 7)  > (even? 'a)  
#t   Error   Error
> (= 5 6)  > (< 5 6)  > (> 5 6)  
#f   #t   #f
> (= 4.5 4.5 4.5)  > (= 4.5 4.5 4.7)  
#t   #f
> (= 'a 'b)  
Error
Outline

• Language elements:
  – Atoms, S-expressions, lists
• Evaluating expressions
  – Function application
  – Quoting an expression
  – Conditionals
  – Defining functions
• Examples
• Function call semantics & higher-order functions
• More examples and features
Conditional Expressions

• \((\textbf{if} \ b \ e_1 \ e_2)\)
  – Evaluate \(b\). If the value is \textit{not} \#f, evaluate \(e_1\) and this is the value to the expression
  – If \(b\) evaluates to \#f, evaluate \(e_2\) and this is the value of the expression

• \((\textbf{cond} \ (b_1 \ e_1) \ (b_2 \ e_2) \ \ldots \ (b_n \ e_n))\)
  – Evaluate \(b_1\). If \textit{not} \#f, evaluate \(e_1\) and use its value. If \(b_1\) evaluates to \#f, evaluate \(b_2\), etc.
  – If all \(b\) evaluate to \#f: unspecified value for the expression; so, we often have \#t as the last \(b\)
  – Alternative form: \((\textbf{cond} \ (b_1 \ e_1) \ (b_2 \ e_2) \ \ldots \ (\textit{else} \ e_n))\)
Function Definition

> (define (double x) (+ x x))
; no values returned

> (double 7)   > (double 4.4)   > (double '(7))
14           8.8            Error

> (define (mydiff x y) (cond ((= x y) #f) (#t #t)))
; no values returned

> (mydiff 4 5)   > (mydiff 4 4)   > (mydiff '(4) '(4))
#t               #f            ???
Outline

• Language elements:
  – Atoms, S-expressions, lists

• Evaluating expressions
  – Function application
  – Quoting an expression
  – Conditionals
  – Defining functions

• Examples

• Function call semantics & higher-order functions

• More examples and features
Member of a List?

In text file mbr.ss create the following:

; this is a comment
; (mbr x list): is x a member of the list?
(define (mbr x list)
  (cond
   ( (null? list) #f )
   ( #t (cond
      ( (equal? x (car list)) #t )
      ( #t (mbr x (cdr list)) ) )
   )
  )
)

Or we could use just one "cond" ...
Member of a List?

In the interpreter:

```lisp
> (load "mbr.ss")  or ,load mbr.ss
mbr.ss
; no values returned
> (mbr 4 '(5 6 4 7))
#t
> (mbr 8 '(5 6 4 7))
#f
```
Union of Two Lists

(define (uni s1 s2)
  (cond
    ( (null? s1) s2)
    ( (null? s2) s1)
    ( #t (cond
         ( (mbr (car s1) s2) (uni (cdr s1) s2))
         ( #t (cons (car s1) (uni (cdr s1) s2))))))

> (uni '(4) '(2 3))
'(4 2 3)

> (uni '(3 10 12) '(20 10 12 45))
'(3 20 10 12 45)

How about using “if” in mbr and uni?
Removing Duplicates

; x: a sorted list of numbers; remove duplicates ...

(define (unique x)
  (cond
   ( (null? x) x )
   ( (null? (cdr x)) x )
   ( (equal? (car x) (cdr x)) (unique (cdr x)) )
   ( #t (cons (car x) (unique (cdr x))) )
  )
)

> (unique '(2 2 3 4 4 5))
  (2 2 3 4 4 5) ;???
Largest Number in a List

; max number in a non-empty list of numbers
(define (maxlist L)
  (cond
    ( (null? (cdr L)) (car L) )
    ( (> (car L) (maxlist (cdr L))) (car L) )
    ( #t (maxlist (cdr L)) )
  )
)

What is the running time as a function of list size? How can we improve it?
A Different Approach

; max number in a non-empty list of numbers
(define (maxlist L) (mymax (car L) (cdr L)))
(define (mymax x L)
  (cond
    ( (null? L) x )
    ( (> x (car L)) (mymax x (cdr L)) )
    ( #t (mymax (car L) (cdr L)) )
  )
)

What is the running time as a function of list size?
Outline

• Language elements:
  – Atoms, S-expressions, lists

• Evaluating expressions
  – Function application
  – Quoting an expression
  – Conditionals
  – Defining functions

• Examples

• S-expressions

• Function call semantics & higher-order functions

• More examples and features
Data Objects in Scheme

• **Atoms**
  – Numeric constants: 5, 20, -100, 2.788
  – Boolean constants: #t (true) and #f (false)
  – String constants: “hi there”
  – Character constants: \a
  – **Symbols**: f, x, +, *, null?, set!
    • Roughly speaking, equivalent to identifiers in imperative languages
  – Empty list: ( )

• **S-expressions**
  – A list is a special case of an S-expression
S-expressions

• Every atom is an S-expression

• If s1 and s2 are S-expressions, so is \(( s1 \ . \ s2 )\)
  – Essentially, a binary tree: left child is the tree for s1, and right child is the tree for s2
  – Atoms are leaves of the tree
    • \((3 \ . \ 5)\)
    • \(((3 \ . \ 4) \ . \ (5 \ . \ 6))\)
    • \((3 \ . \ (5 \ . \ ()))\)
Primitive Functions for S-expressions

• **car**: unary; produces the S-expression corresponding to the left child of the argument
  – Not defined for atoms
• **cdr**: unary; produces the S-expression corresponding to the right child of the argument
  – Not defined for atoms
• **cons**: binary; produces a new S-expr with left child = 1\textsuperscript{st} arg and right child = 2\textsuperscript{nd} arg
Lists

• Special category of S-expressions

• Recursive definition
  – The empty list ( ) is a list; length is 0
  – If the S-expression Y is a list, the S-expression ( X . Y ) is also a list; length is 1 + length of Y
    • ((3 . 4) . (5 . 6)) is not a list
    • (3 . (5 . ())) is a list, with length 2

• Notation: ( e₁ . ( e₂ . ( ... ( eₙ . ( )) ) ) ) is written as ( e₁ e₂ ... eₙ )
Examples of Lists

• \(( (3 \cdot 4) 5 )\) is \(( (3 \cdot 4) \cdot (5 \cdot ( )) )\)
• \(( (3) (4) 5 )\) is \(( (3 \cdot ( )) \cdot ( (4 \cdot ( )) \cdot (5 \cdot ( ))) )\)
• \((A B C)\) is \((A \cdot (B \cdot (C \cdot () )))\)
• \(((A B) C)\) is \(((A \cdot (B \cdot ( ))) \cdot (C \cdot ( )))\)
• \((A B (C D))\) is \((A \cdot (B \cdot ((C \cdot (D \cdot ( ))) \cdot ( ))) )\)
• \(((A))\) is \(((A \cdot ()) \cdot ( ))\)
• \((A (B \cdot C))\) is \((A \cdot ((B \cdot C) \cdot ( )))\)
Lists

• Another view of lists: a binary tree in which
  – the rightmost leaf is ( )
  – the S-expressions hanging from the rightmost “spine” of the tree are the list elements

• List elements can be atoms, other lists, and general S-expressions
  – ( ( 3 4 ) 5 ( 6 ) ) is a list with 3 elements
  – Thus, lists are heterogeneous: the elements do not have to be of the same type

• Empty list ( ) - has zero elements
  – Operations car and cdr are not defined for an empty list – run-time error
Lists

• **car** for a list produces the first element of the list (the list *head*)
  – e.g. for ( ( A B ) ( C D ) E ) will produce ( A B )

• **cdr** produces the *tail* of the list: a list containing all elements except the first
  – e.g. for ( ( A B ) ( C D ) E ) will produce ( ( C D ) E )

• **cons** adds to the beginning of the list
  – cons of A and ( B C ) is ( A B C )
  – e.g., cons of car of x and cdr of x is x
Outline

• Language elements:
  – Atoms, S-expressions, lists

• Evaluating expressions
  – Function application
  – Quoting an expression
  – Conditionals
  – Defining functions

• Examples

• Function call semantics & higher-order functions

• More examples and features
Semantics of Function Calls

• Consider \((F \ p_1 \ p_2 \ \ldots)\)

• Evaluate \(p_1, p_2, \ldots\) using the current bindings

• “Bind” the resulting values \(v_1, v_2, \ldots\) to the formal parameters \(f_1, f_2, \ldots\) of \(F\)
  – add pairs \((f_1, v_1), (f_2, v_2), \ldots\) to the current set of bindings

• Evaluate the body of \(F\) using the bindings
  – if we see \(p_1\) in the body, we evaluate it to value \(v_1\)

• After coming back from the call, the bindings for \(p_1, p_2, \ldots\) are destroyed
Higher-Order Functions

(define (double x) (+ x x))
(define (twice f x) (f (f x)))
(twice double 2)   Returns 8

(define (mymap f list)
  (if (null? list) list
    (cons (f (car list))
      (mymap f (cdr list)))))
(mymap double '(1 2 3 4 5))   Returns '(2 4 6 8 10)
Higher-Order Functions

(define (double x) (+ x x))
(define (id x) x)
((id double) 11) Returns 22

(define (makelist f n)
  (if (= n 0) '()
    (cons f (makelist f (- n 1)))))

(makelist double 4)
Returns '(procedure double, procedure double, procedure double, procedure double)
Higher-Order Functions

\[(\text{define} \ (\text{newmap} \ x \ \text{list}) \ (\text{if} \ (\text{null?} \ \text{list}) \ \text{list} \ (\text{cons} \ ((\text{car} \ \text{list}) \ x) \ (\text{newmap} \ x \ (\text{cdr} \ \text{list}))))))\]

What does this function do?

\[(\text{newmap 11} \ (\text{makelist} \ \text{double} \ 7))\]

What is the result of this function application?

\[(\text{define} \ (f \ n) \ (\text{newmap} \ n \ (\text{makelist} \ \text{double} \ 5))) \ (\text{twice} \ f \ 9)\]

How about here?
Outline

• Language elements:
  – Atoms, S-expressions, lists

• Evaluating expressions
  – Function application
  – Quoting an expression
  – Conditionals
  – Defining functions

• Examples

• Function call semantics & higher-order functions

• More examples and features
Recursion for Iterating

; Factorial function
(define (fact n)
  (if (= n 0) 1
      (* n (fact (- n 1))))))

Equivalent computation in imperative languages
f := 1;
for (i = 1; i <= n; i++) f := f * i;
Quicksort

Sort list of numbers (for simplicity, no duplicates)

Algorithm:
  – If list is empty, we are done
  – Choose pivot \( n \) (e.g., first element)
  – Partition list into lists A and B with elements \(< n\) in A and elements \(> n\) in B
  – Recursively sort A and B
  – Append sorted lists and \( n \)
Constructing the Two Sublists

(define (ltlist n list)
  (if (null? list) list
    (if (< (car list) n)
      (cons (car list) (ltlist n (cdr list)))
      (ltlist n (cdr list)))))

Similarly we can define function gtlist
Sorting

(define (qsort list)
  (if (null? list) list
      (append
       (qsort (ltlist (car list) (cdr list)))
       (cons (car list) '())
       (qsort (gtlist (car list) (cdr list)))))))

(qsort '(4 3 5 1 6 2 8 7))
Returns '(1 2 3 4 5 6 7 8)
A Few Other Language Features

• \texttt{(lambda (x y ...\) body)} : evaluates to a function
  – \texttt{((lambda (x) (+ x x)) 4)} evaluates to 8
  – \texttt{(define (f x y ...\) body)} is equivalent to
    \texttt{(define f (lambda (x y ...) body))}
  – Comes from the \(\lambda\)-calculus, the theoretical foundation for functional languages (Alonzo Church)

• \texttt{let} bindings – give names to values
  – \texttt{(let ((x 2) (y 3)) (* x y))} produces 6
  – \texttt{(let ((x 2) (y 3)) (let ((x 7) (z (+ x y))) (* z x)))} is 35

• \texttt{(define x expr)} and \texttt{(define (f x y ...\) body)} create global bindings for these names